Generic Implementation of Product Line Components

Dirk Muthig and Thomas Patzke

Fraunhofer Institute Experimental Software Engineering (IESE),
Sauerwiesen 6, D-67661 Kaiserslautern, Germany
{muthig, patzke}@iese.fraunhofer.de

Abstract. An argument pro component-based software development is the idea of constructing software systems by assembling preexisting components instead of redeveloping similar or identical functionality always from scratch. Unfortunately, integrating existing components practically means adaptation and use rather than use only, which makes an ideal component-based development hard to realize in practice. Product line engineering, however, tackles this problem by making components as generic as needed for a particular product family and thus allows component reuse. Such a component covers variabilities and thus its implementation must consider variabilities as well.
In this paper, we describe a process for implementing generic product line components and give an overview of variability mechanisms at the implementation level, illustrated by a running example, a generic test component.

1 Introduction

Most of the software organizations today develop and maintain a set of software products rather than a single product only. These products typically are similar products in the same application domain and thus significantly share common characteristics. In order to make software-related tasks as efficient as possible, these commonalities among products should be exploited systematically.

Component-based development is one such approach whose main idea is to construct software systems by assembling components that realize common functionality instead of redeveloping this functionality always from scratch. Unfortunately, integrating existing components practically means reuse (i.e., adaptation and use) rather than use only, which makes a clean, ideal component-based development hard to realize in practice.

Product line engineering, however, is an approach that tackles this problem by making components systematically as generic as needed for a particular product family and thus allows components to be reused easily within a family context [1]. The key concept of the product line approach is the analysis of common and varying characteristics of the products to be built by an organization. The analysis results are then used to construct components that represent commonalities but at the same time are prepared to accommodate the required

variabilities related to the common concept at a lower level of detail [2]. Being prepared means that a component's implementation considers the variations within a product line in some way. In most cases, there is more than a single way of implementing a variability. As usual, each of several possibilities has its specific advantages, as well as disadvantages. Only a systematic weighting of the different pros and cons should eventually lead to a sound decision of which way to choose for implementing a given variability. In practice, these decisions are implicit decisions that are based on the personal experience of software developers. Hence, the decisions are taken unsystematically. To change this situation, pros and cons of different ways for implementing variabilities (i.e. variability mechanisms), as well as experience in applying them, must be recorded.

In this paper, we start to change this situation by discussing several variability mechanisms and explicitly capturing the potential advantages and disadvantages.

The remainder of the paper is organized as follows: Section 2 extends the component model of component technologies by adding to the component implementation an abstract model of the functionality the implementation realizes. In this section, the generic test component used as running example throughout the paper is introduced. Section 3 is the core of the paper. It gives an overview of variability mechanisms at the implementation level and discusses their pro and cons. Section 4 summarizes the results and characterizes the different mechanisms with respect to product line support and problems in applying it in typical industrial contexts. Finally a general process of implementing product line components, which systematically applies variability mechanisms, is introduced.

2 Component Model

For the major component technologies Microsoft's (D)COM/.NET family, Sun's Java Enterprise Beans (EJBs) and the OMG's CORBA component model (CCM) a component is an executable entity with a well-defined interface. The executable is generated from a component's implementation, which means, in the traditional view, source code [3]. As motivated in the introduction, the need for an additional, more abstract view on a component is required - in general but in particular in a product line context - to enhance the reusability of components [2]. In this paper, the component model defined by the KobrA method (see [4] for details) is introduced by means of a concrete example, a generic test component similar to the JUnit test framework [6].

In order to deploy a component, an implementation of the component is required from which automatically an executable entity can be generated. Thereby, the key idea is that an executable entity can be used without touching it. As motivated above, however, components must often be touched in order to benefit from them. Consequently, the component's implementation has to be changed. Otherwise the fixed component would have to be wrapped or would have to be reimplemented in its entirety just to reuse it in a slightly different context.

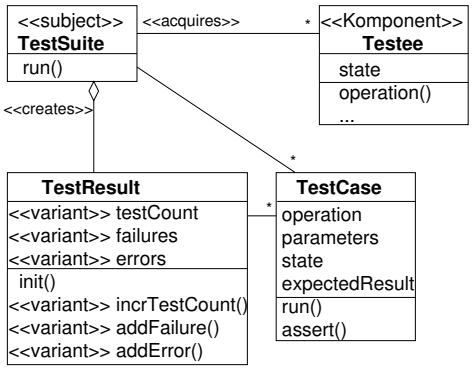


Fig. 1. Generic realization diagram [4] of a test suite

Consider the variant attributes of a TestResult produced by a TestSuite component instance, (see fig. 1 and table 1). One way of implementing these instances with different capabilities is to create separate isolated implementations (see Appendix A).

Table 1. Specification of the TestCase.run() operation

Name	TestCase.run()
Description	Invokes a testee's operation with the defined parameters, analyzes operation output wrt. expected result, and stores test information in testresult
Sends	testee.operation(parameters)
Changes	testresult:TestResult
Assumes	testee is set up correctly and testresult exists
Result	(opt.) testresult.testCount has been incremented, testee.operation(parameters) has been invoked. The operation's result or the testee's state have been retrieved and compared with expectedResult, (opt.) the anticipated (failure) or unanticipated problems in testresult have been updated

The described way of implementing the generic TestSuite component is simple but not scaleable because it results in a combinatorial number of classes with no reuse at all. In order to prevent this, the variabilities of a component must be controlled more systematically also at the implementation level. For that purpose, variability mechanisms are needed.

3 Variability Mechanisms

A variability mechanism [5] is a way of implementing varying characteristics of a component at the implementation level. Goals of variability mechanisms are to minimize code duplication, reuse effort, and maintenance effort.

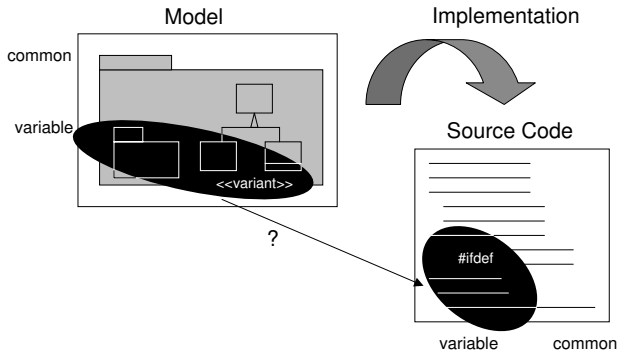


Fig. 2. Implementing variant characteristics

In general, variant characteristics require a generic implementation, which thus contains some variant code. Figure 2 illustrates the mapping of variant elements in the abstract component models to variant parts in the component's implementation.

Before concrete variability mechanisms are described, variabilities are generally classified. Three basic kinds of variabilities can be distinguished: optional and alternative variabilities as well as variabilities offering multiple coexisting possibilities [7].

In the simplest case, only one possibility exists, which can be included or excluded. This optional variability corresponds to a boolean type of variability point. In other cases more than one possibility exists, and all the possibilities are mutually exclusive so that just exactly one can be selected. This alternative variability (XOR) corresponds to an enum-type of variability point. In a third common case, the possibilities are not mutually exclusive so that one or more can be selected; that OR-kind of variability is reflected by a set-type of variability point. By combining these basic kinds of variation point more complex types can be obtained [8].

In the following subsections, traditional and emerging mechanisms for controlling behavioral variabilities of these kinds at the implementation level are explored.

3.1 Conditional Compilation

Variabilities at the conceptual level are physically implemented as optional or alternative code parts (see fig. 2). The problem is to keep all these variable as

well as the common code parts in one place (e.g. file) while at the same time being able to choose one specific configuration. Moreover, one kind of variability often affects several files, whereas the configuration should concentrate in exactly one place.

Macros are one commonly used (often negative) variability mechanism. They represent an early abstraction mechanism introduced in assembler languages. [A macro is] a module containing a body of code that is copied during compilation or assembly when invoked by name. It is not readily modified during invocation ([9], p.347f.).

Macros provide for statically expressing optional and alternative pieces of code at a maximum level of granularity, which means that they can even affect single statements or parts thereof. This fine-graininess is one disadvantage of macros: for large projects, they tend to clutter the code, impeding comprehensibility as a whole. Moreover, conventional macros only have a global scope and, being preprocessing constructs, do not have any language support. They or their parameters might interfere with each other or with constructs of the programming language; they may even be used to change the semantics of the programming language (for example, by redeclaring the keyword private as public). Experience has shown that the use of conventional macros as a product line implementation technology tends to be problematic [10].

As an example, consider the TestCase and TestResult implementations using conditional compilation (Appendix B). Compared to the naive implementation (sec. 2), the number of classes has decreased from 4 to 2. However, the class internals (both the interface, as in TestResultCC, and the implementation, as in TestCase::run()) are cluttered with macros. Moreover, for each new variant (e.g. storing errors) the internals of potentially all classes have to be changed manually, leading to maintenance problems.

Frame technology [9] uses an advanced preprocessor which avoids some of the drawbacks of conventional macros. Components are expressed in hierarchies of frames, which are adaptable components expressed in an arbitrary programming language. A frame contains both program code and frame commands, which serve to add from other frames' capabilities as the application requires. A simple frame processor can be found at [11], and an XML-based one at [12].

3.2 Polymorphism

Subtype Polymorphism Subtype (inclusion) polymorphism is another major common mechanism for handling variabilities. It is implemented in object-oriented languages by defining one abstract class (interface) as a common point of access and implementing the alternative behavior in concrete subclasses; thus it is mainly concerned with the XOR-kind of variability. This mechanism represents one key element of object-oriented design patterns and object-oriented frameworks.

Subtype polymorphism as a product line variability mechanism has at least two negative effects on quality attributes:

First, as subtype polymorphism is applied dynamically (its binding time is run-time), its extra level of indirection appears in the executable code, which has a negative performance impact. In some cases, this run-time flexibility is actually needed as a functional requirement at the same time, for example when one out of several classes is chosen to be instantiated as the result of a run-time event. But the design should explicitly reveal if the binding time has to be run-time, because otherwise a non-run-time mechanism could be more effective.

Second, and worse, subtype polymorphism restricts reusability ([9], p.13). In contrast to generic implementations, it only allows you to effectively add one kind of variability at the same time, for example by subclassing. In other words, variabilities of the OR-kind cannot be cleanly expressed, so that each class represents a feature combination, not a single feature, in its entirety. This results in a huge number of classes and a large amount of code duplication when adding new features. In [15] this phenomenon is referred to as the "tyranny of the dominant decomposition" because one way of composing the system (e.g. into a certain class hierarchy) forces all other features to conform to this composition, even if this restricts their quality attributes (e.g. by duplicating common features). This problem can only be resolved by applying another set of techniques like multiple inheritance, multiple class hierarchies or by applying design patterns like Visitor, with negative impacts on complexity and performance.

With subtype polymorphism TestResult and TestCase can be implemented as sketched in Appendix C. Compared to the implementation with conditional compilation, the number of classes has increased from 2 to 4, which is the number for the naive case. In case of introducing a new variant for storing errors, not all classes have to be changed, and where change is necessary, it is more localized (e.g. for handling errors, the interface of TestResultSP has to be extended, as well as the implementation of TestCaseCC). However, implementing OR-variabilities, e.g. handling both test counts and failures, might lead to code duplication (unless multiple inheritance is used). Moreover, there will be a proliferation of classes as the number of alternatives grows.

Parametric Polymorphism Subtype polymorphism with class inheritance or object composition are two ways of composing behavior in object-oriented systems, and a third one is to employ parametric polymorphism ([13], p.22). It represents an alternative non-OO implementation technique for product lines. As with subtype polymorphism, it allows for implementing common and variable, optional and XOR-variable parts of code, as well as Template and Hook Methods [14]. However, this is not done at run-time, but at construction time,[1] which avoids the aforementioned performance and combinability drawbacks. One the one hand, it allows the compiler to inline the code, so that the extra level of indirection only appears at construction time and so no run-time performance penalties arise. On the other hand, it enables us to implement OR-variabilities in a straightforward way, resulting in a far larger amount of feature combinations

[1] In this paper we do not distinguish between the different kinds of construction time like source-, compile- or link- (load-) time as presented in ([3], p.73).

with less code because whole features (i.e., combinations of incomplete features) can be combined in arbitrary ways, which provides a means for the separation of concerns.

Paul Basset emphasizes this duality between run-time and construction time as well when he defines the latter as 'those times when we vary the data structures and methods that we choose to hold fixed at run-time' ([9], p.13).

Like macros, parametric polymorphism is a static variability mechanism, but it has programming language support and so avoids many of the disadvantages of macros. The drawbacks of parametric polymorphism are that compile-times might increase (which can be a problem if compilations occur frequently), that fewer programming languages support it, and that less experience has been gathered with it.

Whereas in unbounded polymorphism (unconstrained genericity) no constraints on the parameter of variation are expressible, this can be done in bounded parametric polymorphism (constrained genericity). However, this means for explicitly expressing required interfaces is supported by only a few programming languages like ML, Eiffel, or Ada95.

Parametric polymorphism in C++ is dealt with in depth in [16] and [17]. Well-known C++ implementations are the STL [18], Boost [19], FC++ [20], the Lambda Library [21], Blitz++ [22], the MTL [23] and Spirit [24], the latter 3 of which are also generative libraries.

A C++ implementation of TestCase and TestResult using parametric polymorphism is outlined in Appendix D. In this approach, the number of classes can be kept to a minimum, and, if the TestResult classes are parameterized as well, the class proliferation can be avoided.

Other Kinds of Polymorphism In addition to this group of universal polymorphism, the older mechanisms of ad-hoc polymorphism, overloading and casting, can be used as variability mechanisms as well. In combination with object composition, they offer a way to compose behavior in object-oriented systems.

Overloading, emphasizing the procedural paradigm, provides different function implementations sharing the same name, but with different operand types. Casting serves to automatically transfer one type of object into another, enabling black-box adaptation by wrapping.

3.3 Defaults

In all these four polymorphism techniques defaults are a mechanism for explicitly providing one variation point for optional code parts, which is usually done in connection with Null Objects [25].

Null objects offer a default behavior for hook methods, often as an empty operation (In the casting mechanism, the wrapped object can be implemented as a Null Object.). They can either share the same base class as the template method, or they are in a separate inherited class. In the former case, like for default function arguments or default parameterized types, a means of canceling

the commonality by providing a different entity than the default parameter or Default Object serves as a mechanism of negative variability.

3.4 Refinements

Recently emerging approaches combine several of the aforementioned polymorphism techniques with nesting techniques to implement more scalable generic components. In these post-object-oriented approaches, just fragments of classes are encapsulated as components, instead of entire classes. They explicitly leave the variation points open which the aforementioned defaults close. [26] call these emerging mechanisms refinements. They all represent a functionality addition to a program, which introduces a new feature, affecting multiple implementation entities simultaneously. Two broad categories of refinement mechanisms exist: collaboration-based and aspect-oriented mechanisms.

Collaboration-Based Mechanisms Collaboration- and role-based design and implementation approaches were first developed in the 90s ([27,28,29]) and have recently gained increased attention in the area of product lines and object-oriented frameworks ([26,30,31,32]). Whereas conventional object-oriented methods primarily view objects as instantiations of classes, role modeling emphasizes collaborations of objects in which the objects play certain roles. Classes are synthesized by composing the roles their instances play in different collaborations.

A collaboration is a view of an object-oriented design from the perspective of a single concern, service, or feature. Collaborations define the building blocks for application families ([26], p.4).

A recent mechanism for implementing collaboration-based designs is based on the GenVoca model [33], where classes are refined by parameterized inheritance, and several interrelating classes share a common layer, which represents their collaboration. These so-called mixin layers are stacked via parameterized inheritance as well.

Figure 3 shows a refinement consisting of 3 classes and 3 collaborations. Within the collaboration layers, each class represents a role, which is only a partial implementation of the entire class that is created by stacking the collaborations. Note that not all of the roles have to be present; for example if class C does not participate in collaboration 2, role C2 can be omitted.

The layers can be stacked in quasi-arbitrary ways, and even the same collaboration might appear in different layers, so that a large variety of generic component implementations can be produced. Moreover, roles can themselves be implemented as stacks of mixin layers, so that components in different scales of granularity can be implemented using the same technique.

Figure 4 illustrates the mixin-based implementation technique for TestCase and TestResult; the implementation is sketched in Appendix E.

This approach has the following advantages: single concerns like test counting and failure handling are encapsulated in a uniform way, the elements of the dominant decomposition (TestCase and TestResult) are preserved, OR-variations can

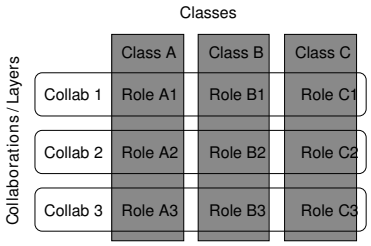


Fig. 3. Collaboration-based design example

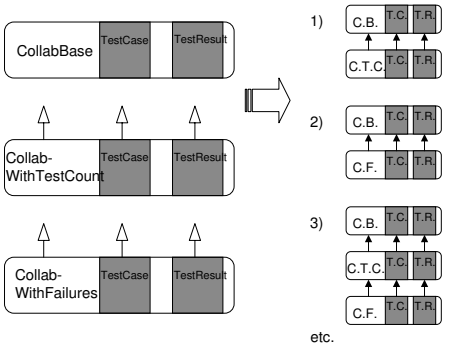


Fig. 4. Mixin-based implementation of the test component

be implemented easily, and new concerns (e.g. error-handling) can be introduced without changing existing code. As an alternative to this inheritance-based implementation, a forwarding based one as presented in ([16], pp.377-379) can be used as well.

Aspect-Orientation A second category of refinements comprises aspect-orientation [15], with several technologies like Aspect-Oriented Programming (AOP), Multi-Dimensional Separation of Concerns (MDSOC), Demeter/AP or Composition Filters. Aspect-Orientation is a separation of concerns technology in which the various crosscutting concerns (properties or areas of interest) of a system and partially their relationships are implemented and then composed automatically, often by an AOP tool. Concerns can be classified into high-level vs. low-level, functional vs. non-functional or development vs. production. Examples for functional concerns are data concerns, feature concerns, business rule concerns or variant concerns. Nonfunctional concerns comprise synchronization, distribution, error handling or transaction management. Among development concerns we find debugging, tracing, testing or profiling; product concerns include user-visible functionality. In the original AOP technique, these properties crosscutting several functional components are captured in modular units called aspects. In order to instantiate the complete system these are combined by the Aspect Weaver, which is a programming-language-dependent tool. In Multi-Dimensional Separation of Concerns, aspects do not only augment classes but other aspects as well. In addition, it provides for adapting existing systems. Its tools allow a developer to compose a collection of separate models, called hyperslices, each encapsulating a concern by defining and implementing a (partial) class hierarchy appropriate for that concern. Further information about aspect-oriented software development can be found at [34].

Aspect-oriented programming, however, is not restricted to a specific tool; it can just as well be applied using common programming languages [35]. Appendix F shows how the test component can be implemented in C++ with aspect-

orientation. This implementation is similar to the collaboration-based one; using namespaces to implement components in different levels of scale. It also offers similar benefits, except that single concerns (test counting, failure handling) are not explicitly encapsulated, and aspect combinations (like both test counting and failure handling) are harder to implement.

3.5 Other Mechanisms

Other variability mechanisms are extension as expressed in the Extension Object [36] or Decorator design patterns, Active Libraries ([37]), reflection and domain-specific languages.

4 Analysis and Outlook

In the previous section we discussed a set of traditional and emerging mechanisms for implementing variabilities. For each of these mechanisms, their specific advantages, as well as disadvantages, have been presented. Table 2 summarizes these results.

Table 2. Advantages and disadvantages of variability mechanisms

Mechanism	Advantages	Disadvantages
Conditional Compilation	well-known, no space/performance loss, fine-gained	no direct language support, unscalable
Subtype Polymorphism	often well-known, dynamic	performance loss, OR-variability difficult
Parametric Polymorphism	no performance loss, all 3 kinds of variability easily expressible, built-in	less known, less supported
Ad-Hoc Polymorphism	often well-known	less important than universal polymorphism
Collaborations	good support for PL implementation	generally not well-known, might require tools
Aspect-Orientation	good support for PL implementation	often requires tools, market caution
Frame Technology	good support for PL implementation	not well-known, requires tool support

All of the mechanisms provide some support for the implementation of product lines but support of different quality. Each mechanism must be seen in the tension of two dimensions: coverage of variability types and complexity. The first dimension characterizes the quality of a mechanism in implementing variabilities. The second dimension characterizes its suitability for applying it in

practice for development and maintenance tasks performed by typical software engineers. Both dimensions are important while planning the systematic introduction of variability mechanisms into a running organization. Figure 5 ranks the variability mechanisms from the previous section with respect to these two dimensions. During a technology transfer project both coverage and complexity are successively increased. An evolutionary migration towards a mature product line implementation approach would thus typically introduce the mechanisms from left to right, or respectively from bottom to top.

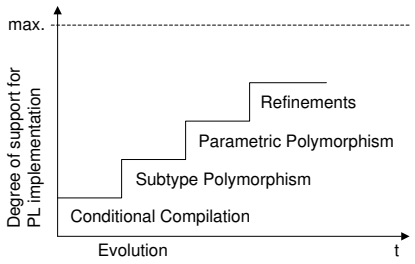


Fig. 5. Variability mechanisms offering different degrees of support for product line implementation

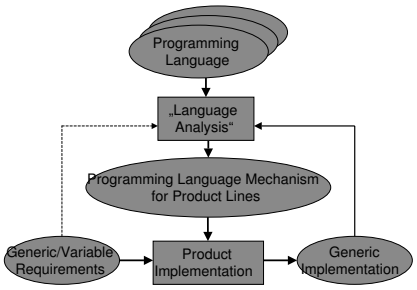


Fig. 6. Process for introducing programming language mechanisms for product lines

The advantages and disadvantages of the different mechanisms are derived from the literature available today and thus represent the state-of-the-art. However, they do not enable a fully systematic selection of the most suitable mechanisms for implementing a given variability. Further experience in applying these mechanisms must be gained and more detailed analysis of the mechanisms are required. New knowledge on variability mechanisms must be made immediately available to software engineers. That is, to learn about variability mechanisms must be integrated into the overall implementation process. Figure 6 captures the initial version of such a process that we have developed in a running research project.

This process starts with an analysis of the variability mechanisms provided by the used programming language and relates these mechanisms to typical variabilities of the particular application domain (a C++ independent version of Coplien's Multi-Paradigm Design [3]). The result of this analysis is a set of patterns for implementing variabilities. Each pattern describes the covered variability types, useful application contexts, and the technical implementation mechanisms. During the implementation, these patterns are applied depending on the variabilities to be realized. Experience from these applications is directly fed back to the language analysis and may cause a change of some variability patterns.

References

1. J. Bayer, O. Flege, P. Knauber, R. Laqua, D. Muthig, K. Schmid, T. Widen, J.-M. DeBaud: PuLSE: A Methodology to Develop Software Product Lines. In Proceedings of the Symposium on Software Reuse, 1999
2. C. Atkinson, D. Muthig: Enhancing Component Reusability through Product Line Technology. Proceedings of the 7th International Conference in Software Reuse (ICSR 02), Springer Press, 2002
3. J. O. Coplien: Multi-Paradigm Design for C++. Addison-Wesley, 1999
4. C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wüst, J. Zettel: Component-Based Product-Line Engineering with UML. Addison-Wesley, 2001
5. I. Jacobson, M. Griss, P. Jonsson: Software Reuse: Architecture, Process and Organization for Business Success. Addison-Wesley, 1997
6. JUnit unit testing Framework: `http://www.junit.org`
7. G. van Gurp, J. Bosch, M. Svahnberg: On the Notion of Variability in Software Product Lines. In Proceedings of Working IEEE/IFIP Conference on Software Architecture (WICSA), 2001
8. D. Muthig: A Light-Weight Approach Facilitating an Evolutionary Transition Towards Software Product Lines. PhD Thesis, University of Kaiserslautern, 2002
9. P. G. Basset: Framing Software Reuse: Lessons from the Real World. Prentice-Hall, 1996
10. M. Anastasopoulos, C. Gacek: Implementing Product Line Variabilities. Proceedings of the 2001 Symposium on Software Reusability (SSR'01), Toronto, Canada, 2001
11. FrameProcessor homepage: `http://frameprocessor.sourceforge.net`
12. XVCL frame processor: `www.comp.nus.edu.sg/labs/software/xvcl.html`
13. E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995
14. M. Fontoura, W. Pree, B. Rumpe: The UML-F Profile for Framework Architectures. Addison-Wesley, 2001
15. Aspect-Oriented Programming. Communications of the ACM Vol.44 No.10, October 2001
16. K. Czarnecki, U. W. Eisenecker: Generative Programming: Methods, Tools and Applications. Addison-Wesley, 2000
17. A. Alexandrescu: Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley, 2001
18. D. R. Musser, A Saini: STL Tutorial and Reference Guide. Addison-Wesley, 1996
19. Boost Library: `http://www.boost.org`
20. FC++ Library: `http://www.cc.gatech.edu/~yannis/fc++`
21. Lambda Library: `http://lambda.cs.utu.fi`
22. Blitz++ Library: `http://www.oonumerics.org.blitz`
23. Matrix Template Library: `http://www.osl.iu.edu/reaearch/mtl`
24. The Spirit Parser Library: `http://spirit.sourceforge.net`
25. B. Woolf. The Null Object. In R. C. Martin, D. Riehle, F. Buschmann (eds.): Pattern Language of Program Design 3. Addison-Wesley, 1998
26. Y. Smaragdakis, D. Batory: Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. To appear in ACM Transactions on Software Engineering and Methodology

27. W. Cunningham, K. Beck: Construction Abstractions for Object-Oriented Applications. Journal of Object-Oriented Programming, July 1989
28. T. Reenskaug: Working with Objects: The OOram Software Engineering Method. Manning Publications, 1996
29. M. VanHilst: Role-Oriented Programming for Software Evolution. PhD Thesis, University of Washington, Seattle, 1997
30. D. Batory, R. Cardone, Y. Smaragdakis: Object-Oriented Frameworks and Product-Lines. 1st Software Product Line Conference, 1999
31. Y. Smaragdakis: Implementing Large-Scale Object-Oriented Components. PhD Thesis, University of Texas, Austin, 1999
32. D. Riehle: Framework Design: A Role Modeling Approach. PhD Thesis, ETH Zürich, 2000
33. D. Batory, B. J. Gerraci: Composition Validation and Subjectivity in GenVoca Generators. IEEE Transactions on Software Engineering, February 1997
34. Aspect-Oriented Software Development: `http://aosd.net`
35. K. Czarnecki, L. Dominick, U. W. Eisenecker: Aspektorientierte Programmierung in C++. iX Magazin, 8-10/2001
36. E. Gamma. Extension Object. In R. C. Martin, D. Riehle, F. Buschmann (eds.): Pattern Language of Program Design 3. Addison-Wesley, 1998
37. K. Czarnecki, U. W. Eisenecker, R. Glück, D. Vandervoorde, T. Veldhuizen: Generative Programming and Active Libraries. Proceedings of the Dagstuhl Seminar 98171 on Generic Programming, LNCS 1766, Springer-Verlag, 2000

Appendix: C++ Implementations

A Naive Implementation

```
class TestResultWithTestCount { // stores the number of tests run
public:
  void incrementTestCount() {...}
};
class TestCaseWithTestCount { // counts the number of tests run
public:
  void run(TestResultWithTestCount& tr) {
    tr.incrementTestCount();
    // perform test
  }
};
class TestResultWithFailures { // stores the detected failures
public:
  void addFailure(...) {...}
};
class TestCaseWithFailures { // considers failures
public:
  void run(TestResultWithFailures& tr) {
    // perform test
    if(failureDetected) tr.addFailure(...);
```

```
  }
};
void main() {
  TestCaseWithTestCount tc; TestResultWithTestCount trc; tc.run(trc);
  TestCaseWithFailures  tf; TestResultWithFailures trf;  tf.run(trf);
}
```

B Conditional Compilation

```
class TestResultCC { // TestResult, conditional compilation
public:
#ifdef WITH_TEST_COUNT
  void incrementTestCount() {...}
#endif
#ifdef WITH_FAILURES
  void addFailure(...) {...}
#endif
};
class TestCaseCC { // TestCase, conditional compilation
public:
  void run(TestResultCC& tr) {
#ifdef WITH_TEST_COUNT
    tr.incrementTestCount();
#endif
    // perform test
#ifdef WITH_FAILURES
    if(failureDetected) tr.addFailure(...);
#endif
  }
};
void main() {
#define WITH_TEST_COUNT
  TestCaseCC tc; TestResultCC trc; tc.run(trc);
#undef WITH_TEST_COUNT
#define WITH_FAILURES
  TestCaseCC tf; TestResultCC trf; tf.run(trf);
#undef WITH_FAILURES
}
```

C Subtype Polymorphism

```
class TestResultSP { // TestResult NullObject
public:
  virtual void incrementTestCount() {}
  virtual void addFailure(...) {}
```

```
};
class TestResultSPWithTestCount: public TestResultSP {
public:
  void incrementTestCount() {...}
};
class TestResultSPWithFailures: public TestResultSP {
public:
  void addFailure(...) {...}
};
class TestCaseSP { // TestCase using subtype polymorphism
public:
  void run(TestResultSP& tr) {
    tr.incrementTestCount();
    // perform test
    if(failureDetected) tr.addFailure(...);
  }
};
void main() {
  TestCaseSP tc;
  TestResultSPWithTestCount trc; tc.run(trc);
  TestResultSPWithFailures  trf; tc.run(trf);
}
```

D Parametric Polymorphism

```
class TestResultPP { // static interface for TestResults
public:
  void incrementTestCount() {}
  void addFailure(...) {}
};
class TestResultPPWithTestCount: public TestResultPP {
public:
  void incrementTestCount() {...}
};
class TestResultPPWithFailures: public TestResultPP {
public:
  void addFailure(...) {...}
};
template<class TESTRESULT>
class TestCasePP { // TestCase using parametric polymorphism
public:
  void run(TESTRESULT& tr) {
    tr.incrementTestCount();
    // perform test
    if(failureDetected) tr.addFailure(...);
```

```
  }
};
void main() {
  TestCasePP<TestResultPPWithTestCount> tc;
  TestResultPPWithTestCount trc; tc.run(trc);

  TestCasePP<TestResultPPWithFailures> tf;
  TestResultPPWithFailures trf; tc.run(trf);
}
```

E Collaboration-Based Mechanism

```
class CollabBase { // static interfaces for TestCase and TestResult
public:
  class TestResult {};
  class TestCase {
  public:
    void run(TestResult&) {}
  };
};
template<class SUBCOLLAB=CollabBase>
class CollabWithTestCount: public SUBCOLLAB {
public:
  class TestResult: public SUBCOLLAB::TestResult {
  public:
    void incrementTestCount() {...}
  };
  class TestCase: public SUBCOLLAB::TestCase {
  typedef typename SUBCOLLAB::TestCase Base;
  public:
    void run(TestResult& tr) {
      tr.incrementTestCount();
      Base::run(tr);
    }
  };
};
template<class SUBCOLLAB=CollabBase>
class CollabWithFailures: public SUBCOLLAB {
public:
  class TestResult: public SUBCOLLAB::TestResult {
  public:
    void addFailure(...) {...}
  };
  class TestCase: public SUBCOLLAB::TestCase {
  typedef typename SUBCOLLAB::TestCase Base;
```

```
  public:
    void run(TestResult& tr) {
      Base::run(tr);
      if(failureDetected) tr.addFailure(...);
    }
  };
};
void main() {
  CollabWithTestCount<>::TestCase tc;
  CollabWithTestCount<>::TestResult trc; tc.run(trc);

  CollabWithFailures<>::TestCase tf;
  CollabWithFailures<>::TestResult trf;  tf.run(trf);

  // both can be combined
  CollabWithFailures<CollabWithTestCount<> >::TestCase tcf;
  CollabWithFailures<CollabWithTestCount<> >::TestResult r;
  tcf.run(r);
}
```

F Aspect-Orientation

```
namespace original {
  class TestResult {};
  class TestCase {
  public:
    void run(TestResult&) { // perform test }
  };
}
namespace aspects {
  typedef original::TestResult TestResult;
  typedef original::TestCase   TestCase;

  class TestResultWithTestCount: public TestResult {
  public:
    void incrementTestCount() {...}
  };
  class TestCaseWithTestCount: public TestCase {
  public:
    void run(TestResultWithTestCount& tr) {
      tr.incrementTestCount();
      TestCase::run(tr);
    }
  };
  class TestResultWithFailures: public TestResult {
```

```
  public:
    void addFailure(...) {...}
  };
  class TestCaseWithFailures: public TestCase {
  public:
    void run(TestResultWithTestCount& tr) {
      TestCase::run(tr);
      if(failureDetected) tr.addFailure(...);
    }
  };
}
void main() {
  aspects::TestCaseWithTestCount tc;
  aspects::TestResultWithTestCount trc;
  tc.run(trc);

  aspects::TestCaseWithFailures tf;
  aspects::TestResultWithFailures trf;
  tf.run(trf);
}
```